The Embedded I/O Company

TEWS TECHNOLOGIES

CARRIER-SW-42

VxWorks Device Driver

IPAC Carrier

Version 3.0.x

User Manual

Issue 3.0.0

December 2011

TEWS TECHNOLOGIES GmbH
Am Bahnhof 7 25469 Halstenbek, Germany
Phone: +49 (0) 4101 4058 0 Fax: +49 (0) 4101 4058 19
e-mail: info@tews.com www.tews.com

CARRIER-SW-42

VxWorks Device Driver

IPAC Carrier

Supported Modules:

TPCI100
TPCI200
TCP201
TCP211
TCP212
TCP213
TCP220
TVME200
TVME201
TVME210
TVME211
TVME220
TVME230
PCI40
CPCI100/200

This document contains information, which is proprietary to TEWS TECHNOLOGIES GmbH. Any reproduction without written permission is forbidden.

TEWS TECHNOLOGIES GmbH has made any effort to ensure that this manual is accurate and complete. However TEWS TECHNOLOGIES GmbH reserves the right to change the product described in this document at any time without notice.

TEWS TECHNOLOGIES GmbH is not liable for any damage arising out of the application or use of the device described herein.

©2007-2011 by TEWS TECHNOLOGIES GmbH

Issue	Description	Date
1.0.0	First Issue	September 16, 2005
1.1.0	Structure definition (ipac_resource) and example applications modified	November 22, 2005
1.2.0	SBS TECHNOLOGIES Carrier Card Support	January 11, 2006
1.2.1	Support of custom carrier boards	April 10, 2006
1.2.2	New Address TEWS TECHNOLOGIES LLC ChangeLog.txt added to file list	December 5, 2006
1.3.0	Type of Parameters in ipFindDevice() changed	June26, 2007
2.0.0	VxBus and SMP support	January 27, 2010
2.0.1	Legacy vs. VxBus Driver modified	March 26, 2010
3.0.0	VxWorks 64-Bit Support added	December 5, 2011

Table of Contents

1 INTRODUCTION 4
2 INSTALLATION 5
2.1 Legacy vs. VxBus Driver 6
2.2 VxBus Driver Installation 6
2.2.1 Direct BSP Builds 7
2.3 Legacy Driver Installation 8
2.3.1 Include the Device Driver in VxWorks Projects 8
2.3.2 Special Installation for Intel x86 based Targets 8
2.3.3 BSP Dependent Adjustments 9
3 CONFIGURATION 10
3.1 VME Bus Carrier Board Setup 10
3.2 Auto Interrupt enable Facility 11
4 INTERFACE FUNCTIONS 12
4.1 ipCarrierInit 12
4.2 ipCarrierPciInit 13
4.3 ipFindDevice 14
4.4 ipFreeDevice 19
4.5 ipac_map_space 20
4.6 ipac_request_irq 22
4.7 ipac_free_irq 24
4.8 ipac_interrupt_ack 26
4.9 ipac_read_uchar 28
4.10 ipac_read_ushort 29
4.11 ipac_read_ulong 30
4.12 ipac_write_uchar 31
4.13 ipac_write_ushort 32
4.14 ipac_write_ulong 33
4.15 ipCarrierShow 34

1 Introduction

IndustryPack (IPAC) carrier boards have different implementations of the system to IndustryPack bus bridge logic, different implementations of interrupt and error handling and so on. Also the different byte ordering (big-endian versus little-endian) of CPU boards will cause problems on accessing the IndustryPack I/O and memory spaces.

To simplify the implementation of IPAC device drivers which work with any supported carrier board, TEWS TECHNOLOGIES has designed a so called carrier driver that hides all of these carrier board differences under a well defined interface.

During the initialization phase the carrier driver will collect information of supported carrier boards and plugged IPAC modules in an internal data base. The data base contains address information for all IP spaces (IO, ID and MEM), the corresponding interrupt vector and level and additional information to identify plugged IP modules and the underlying carrier board.

All resource information necessary for IPAC module driver initialization can be retrieved from this data base by calling the ipFindDevice() function with appropriate arguments to specify the IPAC module we are looking for. If necessary, this function will enable interrupts on the carrier board (e.g. PCI carrier) and related system buses (PCIbus and VMEbus).

Due to the fact that the TEWS TECHNOLOGIES carrier driver and IPAC module drivers are independent, the carrier driver can also be used by custom drivers without any modification.

The CARRIER-SW-42 supports the modules listed below:

TPCI100	PCI carrier for 2 IndustryPack modules
TPCI200	PCI carrier for 4 IndustryPack modules
TCP201	Compact PCI carrier for 4 IndustryPack modules
TCP211	Compact PCI carrier for 2 IndustryPack modules
TCP212	Compact PCI carrier for 2 IndustryPack modules
TCP213	Compact PCI carrier for 2 IndustryPack modules
TCP220	Compact PCI carrier for 4 IndustryPack modules
TVME200	VMEbus carrier for 4 IndustryPack modules
TVME201	VMEbus carrier for 4 IndustryPack modules
TVME210	VMEbus carrier for 2 IndustryPack modules
TVME211	VMEbus carrier for 2 IndustryPack modules
TVME220	VMEbus carrier for 4 IndustryPack modules
TVME230	PCI Expansion Card (SPAN) for 4 IndustryPack modules
PCI40	SBS PCI carrier for 4 IndustryPack modules
CPCI100/200	SBS CompactPCI carrier for 2/4 IndustryPack modules

To get more information about the features and use of supported devices it is recommended to read the manuals listed below.

Carrier Board User Manual
Carrier Board Engineering Manual

2 Installation

Following files are located on the distribution media:

Directory path ‘CARRIER-SW-42’:

CARRIER-SW-42-3.0.0pdf	PDF copy of this manual
CARRIER-SW-42-VXBUS.zip	Zip compressed archive with VxBus driver sources
CARRIER-SW-42-LEGACY.zip	Zip compressed archive with legacy driver sources
ChangeLog.txt	Release history
Release.txt	Release information

The archive CARRIER-SW-42-VXBUS.zip contains the following files and directories:

Directory path ‘./tews/ipac_carrier’:

ipac_carrier_drv.c	Device driver source
ipac_carrier_def.h	Driver include file
ipac_slots.h	Slot descriptions for VME bus carrier boards
include/tvxbHal.h	Hardware dependent interface functions and definitions
export/ipac_carrier.h	Carrier driver interface definitions
Makefile	Driver Makefile
40ipac_carrier.cdf	Component descriptions file for VxWorks development tools
ipac_carrier.dc	Configuration stub file for direct BSP builds
ipac_carrier.dr	Configuration stub file for direct BSP builds

The archive CARRIER-SW-42-LEGACY.zip contains the following files and directories:

Directory path ‘./ipac_carrier’:

carrier_drv.c	Device driver source
carrier_def.h	Driver include file
ipac_slots.h	Slot descriptions for VME bus carrier boards
include/tdhal.h	Hardware dependent interface functions and definitions
export/ipac_carrier.h	Carrier driver interface definitions

2.1 Legacy vs. VxBus Driver

In later VxWorks 6.x releases, the old VxWorks 5.x legacy device driver model was replaced by VxBus-enabled device drivers. Legacy device drivers are tightly coupled with the BSP and the board hardware. The VxBus infrastructure hides all BSP and hardware differences under a well defined interface, which improves the portability and reduces the configuration effort. A further advantage is the improved performance of API calls by using the method interface and bypassing the VxWorks basic I/O interface.

VxBus-enabled device drivers are the preferred driver interface for new developments.

The checklist below will help you to make a decision which driver model is suitable and possible for your application:

Legacy Driver	VxBus Driver
▪ VxWorks 5.x releases ▪ VxWorks 6.5 and earlier releases ▪ VxWorks 6.x releases without VxBus PCI bus support	▪ VxWorks 6.6 and later releases with VxBus PCI bus ▪ SMP systems (only the VxBus driver is SMP safe!) ▪ 64-bit systems (only the VxBus driver is 64-bit compatible)

2.2 VxBus Driver Installation

Because Wind River doesn't provide a standard installation method for 3rd party VxBus device drivers the installation procedure needs to be done manually.

In order to perform a manual installation extract all files from the archive CARRIER-SW-42-VXBUS.zip to the typical 3rd party directory *installDir/vxworks-6.x/target/3rdparty* (whereas *installDir* must be substituted by the VxWorks installation directory).

After successful installation the CARRIER device driver is located in the vendor and driver-specific directory *installDir/vxworks-6.x/target/3rdparty/tews/ipac_carrier.*

At this point the CARRIER driver is not configurable and cannot be included with the kernel configuration tool in a Wind River Workbench project. To make the driver configurable the driver library for the desired processer (CPU) and build tool (TOOL) must be built in the following way:

(1) Open a VxWorks development shell (e.g. C:\WindRiver\wrenv.exe -p vxworks-6.7)

(2) Change into the driver installation directory
installDir/vxworks-6.x/target/3rdparty/tews/ipac_carrier

(3) Invoke the build command for the required processor and build tool with optional VXBUILD argument
make CPU=cpuName TOOL=tool [VXBUILD=xxx]

For Windows hosts this may look like this:

```
> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\ipac_carrier
> make CPU=PENTIUM4 TOOL=diab
```

To compile SMP-enabled libraries, the argument VXBUILD=SMP must be added to the command line

```
> make CPU=PENTIUM4 TOOL=diab VXBUILD=SMP
```

To build 64-bit libraries, the argument VXBUILD=LP64 must be added to the command line

```
> make TOOL=gnu CPU=CORE VXBUILD=LP64
```

For 64-bit SMP-enabled libraries a build command may look like this

```
> make TOOL=gnu CPU=CORE VXBUILD="LP64 SMP"
```

To integrate the CARRIER driver with the VxWorks development tools (Workbench), the component configuration file *40ipac_carrier.cdf* must be copied to the directory *installDir/vxworks-6.x/target/config/comps/VxWorks*.

```
> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\ipac_carrier
> copy 40ipac_carrier.cdf \Windriver\vxworks-6.7\target\config\comps\vxWorks
```

In VxWorks 6.7 and newer releases the kernel configuration tool scans the CDF file automatically and updates the *CxrCat.txt* cache file to provide component parameter information for the kernel configuration tool as long as the timestamp of the copied CDF file is newer than the one of the *CxrCat.txt*. If your copy command preserves the timestamp, force to update the timestamp by a utility, such as *touch*.

In earlier VxWorks releases the CxrCat.txt file may not be updated automatically. In this case, remove or rename the original *CxrCat.txt* file and invoke the make command to force recreation of this file.

```
> cd \Windriver\vxworks-6.7\target\config\comps\vxWorks
> del CxrCat.txt
> make
```

After successful completion of all steps above and restart of the Wind River Workbench, the CARRIER driver can be included in VxWorks projects by selecting the *"TEWS IPAC CARRIER Driver"* component in the *"hardware (default) - Device Drivers"* folder with the kernel configuration tool.

2.2.1 Direct BSP Builds

In development scenarios with the direct BSP build method without using the Workbench or the vxprj command-line utility, the CARRIER configuration stub files must be copied to the directory *installDir/vxworks-6.x/target/config/comps/src/hwif.* Afterwards the *vxbUsrCmdLine.c* file must be updated by invoking the appropriate make command.

```
> cd \WindRiver\vxworks-6.7\target\3rdparty\tews\ipac_carrier
> copy ipac_carrier.dc \Windriver\vxworks-6.7\target\config\comps\src\hwif
> copy ipac_carrier.dr \Windriver\vxworks-6.7\target\config\comps\src\hwif

> cd \Windriver\vxworks-6.7\target\config\comps\src\hwif
> make vxbUsrCmdLine.c
```

2.3 Legacy Driver Installation

2.3.1 Include the Device Driver in VxWorks Projects

In order to include the CARRIER-SW-42 device driver into a VxWorks project (e.g. Tornado IDE or Workbench) follow the steps below:

(1) Extract all files from the archive CARRIER-SW-42-LEGACY.zip to your project directory.

(2) Add the device drivers C-files to your project.

(3) Now the driver is included in the project and will be built with the project.

For a more detailed description of the project facility please refer to your VxWorks User's Guide (e.g. Tornado, Workbench, etc.)

2.3.2 Special Installation for Intel x86 based Targets

The CARRIER device driver is fully adapted for Intel x86 based targets. This is done by conditional compilation directives inside the source code and controlled by the VxWorks global defined macro **CPU_FAMILY**. If the content of this macro is equal to *I80X86* special Intel x86 conforming code and function calls will be included.

The second problem for Intel x86 based platforms can't be solved by conditional compilation directives. Due to the fact that some Intel x86 BSP's doesn't map PCI memory spaces of devices which are not used by the BSP, the required device memory spaces can't be accessed.

To solve this problem a MMU mapping entry has to be added for the required carrier board PCI memory spaces prior the MMU initialization (*usrMmuInit()*) is done.

The function ipCarrierPciInit() will add MMU table entries for all used PCI address spaces on supported (compact)PCI carrier boards. Please insert a call to this function after the PCI initialization is done and prior to MMU initialization (*usrMmuInit()*).

The right place to call the function *ipCarrierPciInit()* is at the end of the function *sysHwInit()* in **sysLib.c** (it can be opened from the project *Files* window).

Be sure that the function is called prior to MMU initialization otherwise the carrier board PCI spaces remains unmapped and an access fault occurs during driver initialization.

Please insert the following call at a suitable place in **sysLib.c**:

```
ipCarrierPciInit();
```

Modifying the sysLib.c file will change the sysLib.c in the BSP path. Remember this for future projects and recompilations.

Because the number of free MMU table entries is limited, an error could occur and not all spaces were mapped. In case of a mapping error the function ipFindDevice() will return with ERROR and a detailed error description will appear on the console. To solve this problem, MMU entries must be added manually to the MMU table in sysLib.c. Edit the file sysLib.c and search for the macro *DUMMY_MMU_ENTRY* in the array *sysPhysMemDesc*. Now you can add new entries by copy and paste of an existing *DUMMY_MMU_ENTRY* entry. Each TEWS TECHNOLOGIES (compact)PCI carrier board requires 5 entries.

2.3.3 BSP Dependent Adjustments

The driver includes a file called *include/tdhal.h* which contains functions and definitions for BSP adaptation. It may be necessary to modify them for BSP specific settings. Most settings can be made automatically by conditional compilation set by the BSP header files, but some settings must be configured manually. There are two way of modification, first you can change the *include/tdhal.h* and define the corresponding definition and its value, or you can do it, using the command line option *–D*.

There are 3 offset definitions (*USERDEFINED_MEM_OFFSET*, *USERDEFINED_IO_OFFSET*, and *USERDEFINED_LEV2VEC*) that must be configured if a corresponding warning message appears during compilation. These definitions always need values. Definition values can be assigned by command line option *-D<definition>=<value>*.

Definition	Description
USERDEFINED_MEM_OFFSET	The value of this definition must be set to the offset between CPU-Bus and PCI-Bus Address for PCI memory space access
USERDEFINED_IO_OFFSET	The value of this definition must be set to the offset between CPU-Bus and PCI-Bus Address for PCI I/O space access
USERDEFINED_LEV2VEC	The value of this definition must be set to the difference of the interrupt vector (used to connect the ISR) and the interrupt level (stored to the PCI header)

Another definition allows a simple adaptation for BSPs that utilize a *pciIntConnect()* function to connect shared (PCI) interrupts. If this function is defined in the used BSP, the definition of *USERDEFINED_SEL_PCIINTCONNECT* should be enabled. The definition by command line option is made by *-D<definition>.*

Please refer to the BSP documentation and header files to get information about the interrupt connection function and the required offset values.

3 Configuration

3.1 VME Bus Carrier Board Setup

Due to the fact that the VME bus isn't a Plug&Play bus, VME bus resources (memory, interrupts, etc.) must be configured manually.

The header file ipac_slots.h in the carrier driver directory contains a dynamically expandable array of type "*struct carrier_slot_desc*". Each array item must be filled with resource information of a single IPAC slot. For a 4-slot IP carrier board (e.g. TVME200), 4 slot entries must be added. The maximum number of slot entries is only limited by the system memory.

To terminate the array, a slot entry with slotIndex = -1 must be added at the end of the array.

```
static struct carrier_slot_desc {
        int             slotIndex;
        unsigned long   ioBase;
        unsigned long   idBase;
        unsigned long   memBase;
        int             intVec;
        int             int0Lvl;
        int             int1Lvl;
};
```

slotIndex

Specifies the slot on the carrier board (slot A = 0, slot B = 1 and so on or -1 for end of list).

ioBase

Mapped address of the IPAC IO space as seen from the CPU (usually not the real VME address). For example if the VME Bus A16/D16 address window appears at CPU address 0xF1FF0000 and the carrier board slot IO space is configured to A16/D16 address 0x6000, ioBase must be set to 0xF1FF6000. Please refer to the BSP documentation about mapping of VME bus spaces.

idBase

Mapped address of the IPAC ID space as seen from the CPU (see also ioBase).

memBase

Mapped address of the IPAC MEM space as seen from the CPU (see also ioBase). Usually this space appears in the VME bus A24 or A32 address space.

intVec

VME bus interrupt vector used by this slot respective plugged IPAC module. Information of free useable VME bus interrupt vectors should be found in the BSP user manual.

int0Lvl

VME bus interrupt level (1..7) for IPAC INTREQ0#. The level must match the configuration of the carrier board. This level is used by the carrier driver to enable the VME bus interrupt level with sysIntEnable().

int1Lvl

VME bus interrupt level (1..7) for IPAC INTREQ1# (see also int0Lvl).

EXAMPLE

The example below configures a 4 slot carrier (e.g. TVME200 with a TVME8240A CPU board)

```
static struct carrier_slot_desc  slot_desc[] = {

/*                                                      Interrupt     */
/*    slot   IO-Space     ID-Space   MEM-Space    Vector   Level   */
/*    index    base         base       base              INT0 INT1 */
/*   ------------------------------------------------------------------*/
    {   0, 0xF1FF6000,  0xF1FF6080, 0xF0D00000,   0xA0,   1,   2 },
    {   1, 0xF1FF6100,  0xF1FF6180, 0xF0D40000,   0xA4,   3,   4 },
    {   2, 0xF1FF6200,  0xF1FF6280, 0xF0D80000,   0xA8,   5,   6 },
    {   3, 0xF1FF6300,  0xF1FF6380, 0xF0DC0000,   0xAC,   7,   0 },

    /*   Please add slot entries here! */


    /* end of list entry (slot_index must be -1) */
    {  -1, 0, 0, 0, 0, 0, 0 },
};
```

See also the comments in ipac_slots.h for a detailed description of this configuration example.

By default this example configuration is disabled by conditional compilation. To enable the configuration the macro EXAMPLE must be defined, either inside ipac_slots.h (#define EXAMPLE) or at the build command line (-DEXAMPLE).

3.2 Auto Interrupt enable Facility

By default bus related interrupt levels (PCI bus, VME bus) will be enabled automatically if the IPAC module driver requires interrupt handling.

The following macros in the driver source file (ipac_carrier_drv.c respective carrier_drv.c) can be defined or not to control the auto interrupt level enable facility.

AUTO_PCI_INT_ENABLE

Define this macro to enable PCI interrupt level by the carrier driver.

AUTO_VME_INT_ENABLE

Define this macro to enable VME interrupt level by the carrier driver.

For TEWS TECHNOLOGIES IPAC module drivers the interrupts must be enabled automatically.

4 Interface Functions

4.1 ipCarrierInit

NAME

ipCarrierInit - IPAC carrier driver initialization

SYNOPSIS

#include “ipac_carrier.h”

STATUS ipCarrierInit(void)

DESCRIPTION

For the legacy carrier driver this function must be called before the first call to ipFindDevice(). For the VxBus carrier driver the initialization function is called automatically from the VxBus subsystem during startup.

During carrier driver initialization, the peripheral busses will be scanned for supported carrier boards. All found slots on PnP (PCI) carrier boards and manually configured slots (VME bus) will be added to an internal data base. In a second phase, the carrier driver will check every slot for mounted IPAC modules. All collected information will now be available for the ipFindDevice() function.

Calling the ipCarrierInit() function is mandatory for the legacy carrier driver and unnecessary for the VxBus carrier driver. For compatibility purposes this function is also available (dummy) for the VxBus carrier driver.

EXAMPLE

```
#include "ipac_carrier.h”

/*
**  First initialize the IP carrier driver if not already done.
**  Note: ipCarrierInit() can be called several times.
*/
if (ipCarrierInit() == ERROR) {
     printf("ERROR: IPAC carrier driver initialization failed\n");
}
```

RETURNS

OK if initialization was successful or ERROR if not.

4.2 ipCarrierPciInit

NAME

ipCarrierPciInit – Generic PCI device initialization

SYNOPSIS

void ipCarrierPciInit()

DESCRIPTION

This function is only required for Intel x86 VxWorks platforms (see also 2.3.2). The purpose is to setup the MMU mapping for all required carrier board PCI spaces (base address register) on supported (Compact)PCI carrier boards.

The right place to call the function *ipCarrierPciInit()* is at the end of the function *sysHwInit()* in **sysLib.c** (it can be opened from the project *Files* window).

This function is only declared and necessary for legacy carrier driver.

EXAMPLE

```
extern void ipCarrierPciInit();

ipCarrierPciInit();
```

4.3 ipFindDevice

NAME

ipFindDevice – Find the specified IPAC module on supported carrier boards

SYNOPSIS

```
STATUS ipFindDevice
(
        unsigned long       manufacturerID,
        unsigned long       modelNumber,
        int                 index,
        unsigned long       slotConfig,
        struct ipac_resource  *ipac
)
```

DESCRIPTION

This function searches for the IPAC module specified by the manufacturer ID, the IPAC model number and the sequence index in the internal database. If the specified module was found (return value OK), the carrier slot will be configured as specified in the argument slotConfig and the structure ipac_resource will be filled with information required to setup the appropriate IPAC module driver.

The argument index specifies the sequence number if more than one module of the same type is installed on supported carrier boards. To select the first module, index must be set to 0, for the second module set index to 1 and so on.

The sequence of IPAC modules is always deterministic. Usually the PCI bus will be searched from lower buses to higher buses and from lower devices to higher devices. On carrier boards the slots will be enumerated from lower slots to higher slots. For VME bus carrier boards the sequence index is given by the setup of the slot description array in ipac_slots.h.

The configuration for the carrier slot where the allocated IPAC module is installed is passed by the argument slotConfig to the carrier driver. More than one configuration items can be combined by using a bit-wise OR.

PARAMETER

manufacturerID

Specifies an 8-bit (IDPROM Data Format I) or 24-bit (IDPROM Data Format II) board manufacturer ID (e.g. 0xB3 for TEWS TECHNOLOGIES).

modelNumber

Specifies an 8-bit (IDPROM Data Format I) or 24-bit (IDPROM Data Format II) model number (manufacturer specific).

index

Sequence index to select a certain IPAC module if more than one module of the same type is installed.

slotConfig

Specifies configuration items for the carrier slot where the IPAC module is installed. The following configuration flags are defined. Use a bit-wise OR to combine more than one flag.

Value	Description
IPAC_INT0_EN	Enable INTREQ0# on the carrier board and the related bus interrupt level. If the auto enable interrupt level feature is enable the CPU board interrupt level will be enabled also with the appropriate system function (intEnable(), sysIntEnable() or vxbIntEnable()).
IPAC_INT1_EN	Same as INTREQ0# for IPAC INTREQ1#
IPAC_EDGE_SENS	Enable edge-sensitive interrupt requests. Only supported by the VxBus enabled carrier driver.
IPAC_LEVEL_SENS	Enable level-sensitive interrupt requests (default)
IPAC_CLK_8MHZ	IPAC clock rate is 8 MHz (default)
IPAC_CLK_32MHZ	IPAC clock rate is 32 MHz
IPAC_MEM_8BIT	The IPAC MEM space is 8-bit wide. Only supported by TEWS PCI carrier boards.
IPAC_MEM_16BIT	The IPAC MEM space is 16-bit wide (default)
IPAC_IACK_CYC	If the IPAC module requires an IACK cycle to acknowledge a pending interrupt, this flag must be set. This configuration is only relevant for PCI carrier boards. If set, the carrier driver will install an ISR which is called before the IPAC module driver ISR is called. This ISR performs a read access to the carrier slot INT space to obtain the interrupt vector from the module (IACK cycle).

ipac

On success this structure is filled with resource information of the slot where the IPAC module is installed. This information can be used to setup the appropriate IPAC module driver.
The memory for the IPAC resource information must be allocated statically (e.g. in the device control block) and must be unique for every IPAC module. Because the IPAC carrier access functions reference the space descriptors within this structure they must be available as long as the IPAC module is referenced by the device driver or application program.

```
struct ipac_resource {
        int                     carrier_type;
        int                     slotIndex;
        unsigned char           *ioBase;
        unsigned char           *idBase;
        unsigned char           *memBase;
        int                     intVec;
        int                     int0Lvl;
        int                     int1Lvl;
        int                     moduleId;
        struct addr_space_desc  idSpace;
        struct addr_space_desc  ioSpace;
        struct addr_space_desc  memSpace;
        unsigned long           slotConfig;
        void                    *pSlotInfo;
        void                    *pVxBusInst;
};
```

carrier_type

Type of carrier board where the IP module is plugged (IPAC_TEWS_PCI, IPAC_SBS_PCI, IPAC_VME …).

slotIndex

Specifies the slot on the carrier board (slot A = 0, slot B = 1…).

ioBase, idBase, memBase

Pointer to the IPAC module IO, ID and MEM space

intVec

Interrupt vector which can be used to connect the module ISR. Only valid for legacy driver PCI bus and VME bus carrier boards.

int0Lvl, int1Lvl

Interrupt level which corresponds to the IPAC module INTREQ0# and INTREQ1#.
Only valid for legacy driver PCI bus and VME bus carrier boards.

moduleId

1:1 copy of the ipCarrierFind() argument *index*.

idSpace, ioSpace, memSpace

Address space descriptor for IPAC module ID, IO and MEM space. Beside the real address an address space descriptor defines the type of access (e.g. endian mode).

EXAMPLE

```
#include ”ipac_carrier.h”

STATUS    result;
struct ipac_resource  ipac1, ipac2, ipac3;


/*
**   Find an IP module from TEWS TECHNOLOGIES (manufacturer = 0xB3)
**   with model number 0x33. This module does not use interrupts and
**   we need only the IO space base address for the related driver.
*/

result = ipFindDevice(0xB3, 0x33, 0, IPAC_CLK_8MHZ, &ipac1);

if (result == ERROR)
{
     printf("ERROR: No IP found\n");
}


/*
**   Find the same module type as above but attach the second module
**   found.
*/

result = ipFindDevice(0xB3, 0x33, 1, IPAC_CLK_8MHZ, &ipac2);

if (result == ERROR)
{
     printf("ERROR: No IP found\n");
}


/*
**   The following module generates level sensitive interrupts at INT0.
**   and has a 16-bit wide memory interface. Important for PCI carrier,
**   this module requires an IACK cycle to acknowledge a pending
**   interrupt.
*/

result = ipFindDevice(  0xB3,
                        0x1C,
                        0,
```

```
                        IPAC_INT0_EN | IPAC_LEVEL_SENS
                        | IPAC_CLK_8MHZ | IPAC_MEM_16BIT | IPAC_IACK_CYC,
                        &ipac3);

if (result == ERROR)
{
     printf("ERROR: No IP found\n");
}
```

RETURNS

OK if the specified IPAC module was found or ERROR if not.

4.4 ipFreeDevice

NAME

ipFreeDevice – Free IPAC module resources

SYNOPSIS

STATUS ipFreeDevice(struct ipac_resource *ipac)

DESCRIPTION

This function returns allocated resources for this IPAC module instance and setup the carrier board slot (only PCI carrier) to a well known inactive state. Before calling this function the IPAC interrupt handling must be disabled by calling the ipac_free_irq() function.

On success the carrier board “in use” count will be decremented to make this VxBus instance removable for hot-plugging purposes (if supported).

This function is only implemented in the VxBus carrier driver.

PARAMETER

ipac

Pointer to IPAC module resource handle that was allocated by a prior call to ipFindDevice().

EXAMPLE

```
#include “ipac_carrier.h”

STATUS    result;
struct ipac_resource  ipac;

result = ipFreeDevice(&ipac)

if (result == ERROR)
{
     printf("ERROR: freeing IPAC device failed\n");
}
```

RETURNS

OK if the specified IPAC device was successful freed ERROR if not.

4.5 ipac_map_space

NAME

ipac_map_space – Obtain an IPAC address space descriptor

SYNOPSIS

struct addr_space_desc *ipac_map_space (struct ipac_resource *ipac, int space_id)

DESCRIPTION

This function returns a space descriptor handle for specified IPAC module space. This handle will be used to access the corresponding IPAC space with the ipac_read_*() and ipac_write_*() access functions.

As long as the returned address space descriptor is used to access the IPAC spaces the referenced *ipac* resource descriptor must be available.

PARAMETER

ipac

Pointer to IPAC module resource handle that was allocated by ipFindDevice().

space_id

Selects the address space. Valid space identifiers are:

Value	Description
IPAC_IO_SPACE	IPAC module IO space
IPAC_ID_SPACE	IPAC module ID space
IPAC_MEM_SPACE	IPAC module MEM space

EXAMPLE

```
#include “ipac_carrier.h”

struct ipac_resource  ipac;
struct addr_space_desc *id_space;


result = ipFindDevice(0xB3, 0x33, 0, IPAC_CLK_8MHZ, &ipac);

/*...*/

if ((id_space = ipac_map_space(ipac, IPAC_ID_SPACE)) == NULL)
{
     printf("mapping ID space failed");
}
```

RETURNS

Returns a pointer to the space descriptor inside the module resource handle or NULL if an error occurred.

4.6 ipac_request_irq

NAME

ipac_request_irq – Connect an interrupt service routine to the module interrupt

SYNOPSIS

```
STATUS ipac_request_irq
(
        struct ipac_resource    *ipac,
        VOIDFUNCPTR             *vector,
        VOIDFUNCPTR             routine,
        long                    parameter
);
```

DESCRIPTION

This function connects an interrupt service routine to the module interrupt and enables the appropriate interrupt level if the auto interrupt enable facility is enabled.

PARAMETER

ipac

Pointer to IPAC module resource handle that was allocated by ipFindDevice().

vector

Interrupt vector to connect. This parameter is not used for VxBus devices.

routine

Pointer to the interrupt service routine to connect.

parameter

Argument that will be passed to the interrupt service routine.

EXAMPLE

```
#include “ipac_carrier.h”

struct ipac_resource  ipac;
STATUS result;
int arg;


result = ipFindDevice(0xB3, 0x33, 0, IPAC_CLK_8MHZ, &ipac);

/*...*/

result = ipac_request_irq(   ipac,
                             INUM_TO_IVEC(ipac.intVec),
                             ISR_func,
                             arg );

if (result == ERROR)
{
     printf(“connectig ISR failed\n”);
}
```

RETURNS

OK on success or ERROR if the ISR cannot be connected.

4.7 ipac_free_irq

NAME

ipac_free_irq – Disconnect an interrupt service routine from the module interrupt

SYNOPSIS

```
STATUS ipac_free_irq
(
        struct ipac_resource    *ipac,
        VOIDFUNCPTR             *vector,
        VOIDFUNCPTR             routine,
        long                    parameter
);
```

DESCRIPTION

This function disconnects an interrupt service routine from the module interrupt and disables the appropriate interrupt level if the auto interrupt enable facility is enabled.

This function is only implemented in the VxBus carrier driver.

PARAMETER

ipac

Pointer to IPAC module resource handle that was allocated by ipFindDevice().

vector

Interrupt vector to disconnect

routine

Pointer to the interrupt service routine to disconnect

parameter

Argument passed to the interrupt service routine.

EXAMPLE

```
#include “ipac_carrier.h”

struct ipac_resource  ipac;
STATUS result;
int arg;


result = ipFindDevice(0xB3, 0x33, 0, IPAC_CLK_8MHZ, &ipac);

/*...*/

result = ipac_request_irq(   ipac,
                             INUM_TO_IVEC(ipac.intVec),
                             ISR_func,
                             arg );

/*...*/

result = ipac_free_irq( ipac,
                        INUM_TO_IVEC(ipac.intVec),
                        ISR_func,
                        arg );

if (result == ERROR)
{
     printf(“connectig ISR failed\n”);
}
```

RETURNS

OK on success or ERROR if the ISR cannot be connected.

4.8 ipac_interrupt_ack

NAME

ipac_interrupt_ack – Acknowledge a pending interrupt and return the interrupt status

SYNOPSIS

STATUS ipac_interrupt_ack(struct ipac_resource *ipac, struct ipac_intstatus *intstatus);

DESCRIPTION

This function performs an IACK cycle by reading the INT space of PCI based carrier boards and returns the vector and status of both IPAC interrupt request lines.

For IPAC module interrupts that are acknowledged by this function the slot configuration flag IPAC_IACK_CYC should not be set, otherwise a pending interrupt may be acknowledged by the auto IACK facility (see also 4.3) before this function is called.

Usually this function will be called within the interrupt service routine for an IPAC module which does not provide a dedicated interrupt status register.

This function is only implemented in the VxBus carrier driver.

PARAMETER

ipac

Pointer to IPAC module resource handle that was allocated by ipFindDevice().

intstatus

Pointer to a variable of type struct ipac_intstatus, which obtains the read vector and interrupt status

```
struct ipac_intstatus {
    int     INT0_active;
    int     INT0_vector;
    int     INT1_active;
    int     INT1_vector;
};
```

INT0_active

TRUE if the IPAC INT0 interrupt is active

INT0_vector

Read interrupt vector (INT space IACK cycle) if INT0 is active and this feature is available.

INT1_active

TRUE if the IPAC INT1 interrupt is active

INT1_vector

Read interrupt vector (INT space IACK cycle) if INT1 is active and this feature is available.

EXAMPLE

```
#include “ipac_carrier.h”

struct ipac_resource  ipac;
struct ipac_intstatus intstatus;
STATUS result;


result = ipac_interrupt_ack(&ipac, &instatus);

if (result == ERROR)
{
     printf(“Feature is not available\n”);
}
```

RETURNS

Returns OK if this feature is available (TEWS PCI carrier boards) or ERROR if not.

4.9 ipac_read_uchar

NAME

ipac_read_uchar – Read one byte (8-bit) from IPAC space

SYNOPSIS

unsigned char ipac_read_uchar(struct addr_space_desc *space, unsigned long offset);

DESCRIPTION

Read one byte (8-bit) from the IPAC space location specified by the address space descriptor and the relative offset.

This access function always expects big-endian IPAC spaces either by hardware design (TVME8xxx CPU boards) or re-programming the BAR layout of TEWS TECHNOLOGIES (Compact)PCI carrier boards. For little-endian SBS (Compact)PCI carrier boards the byte lanes will be swapped.

On PowerPC boards the access will be ordered (EIEIO).

PARAMETER

space

Address space descriptor pointer.

offset

Address offset (bytes) within this space.

EXAMPLE

```
#include ”ipac_carrier.h”

struct addr_space_desc *id_space;
unsigned char modelNumber;

/* ... */

modelNumber = ipac_read_uchar( id_space, 0x0B );
```

RETURNS

Returns the read byte.

4.10 ipac_read_ushort

NAME

ipac_read_ushort – Read one word (16-bit) from IPAC space

SYNOPSIS

unsigned short ipac_read_ushort(struct addr_space_desc *space, unsigned long offset);

DESCRIPTION

Read one word (16-bit) from the IPAC space location specified by the address space descriptor and the relative offset.

This access function always performs a big-endian access. Depending on the CPU architecture and carrier board hardware byte lanes will be swapped as necessary.

On PowerPC boards the access will be ordered (EIEIO).

PARAMETER

space

Address space descriptor pointer.

offset

Address offset (bytes) within this space.

EXAMPLE

```
#include ”ipac_carrier.h”

struct addr_space_desc *io_space;
unsigned short data;

/* ... */

data = ipac_read_ushort( io_space, 0x40 );
```

RETURNS

Returns the read word.

4.11 ipac_read_ulong

NAME

ipac_read_ulong – Read one long word (32-bit) from IPAC space

SYNOPSIS

unsigned long ipac_read_ulong(struct addr_space_desc *space, unsigned long offset);

DESCRIPTION

Read one long word (32-bit) from the IPAC space location specified by the address space descriptor and the relative offset.

This access function always performs a big-endian access. Depending on the CPU architecture and carrier board hardware word byte and word lanes will be swapped as necessary.

On PowerPC boards the access will be ordered (EIEIO).

PARAMETER

space

Address space descriptor pointer.

offset

Address offset (bytes) within this space.

EXAMPLE

```
#include ”ipac_carrier.h”

struct addr_space_desc *mem_space;
unsigned long data;

/* ... */

data = ipac_read_ulong( mem_space, 0x1000 );
```

RETURNS

Returns the read long word.

4.12 ipac_write_uchar

NAME

ipac_write_uchar – Write one byte (8-bit) to an IPAC space

SYNOPSIS

void ipac_write_uchar(struct addr_space_desc *space, unsigned long offset , unsigned char value);

DESCRIPTION

Write one byte (8-bit) to the IPAC space location specified by the address space descriptor and the relative offset.

This access function always expects big-endian IPAC spaces either by hardware design (TVME8xxx CPU boards) or re-programming the BAR layout of TEWS TECHNOLOGIES (Compact)PCI carrier boards. For little-endian SBS (Compact)PCI carrier boards the byte lanes will be swapped.

On PowerPC boards the access will be ordered (EIEIO).

PARAMETER

space

Address space descriptor pointer.

offset

Address offset (bytes) within this space.

value

Data byte (8-bit) to write to the specified location.

EXAMPLE

```
#include ”ipac_carrier.h”

struct addr_space_desc *io_space;

/* ... */

/* write 0x55 to offset 0x20 within the IO space */
ipac_write_uchar( io_space, 0x20, 0x55 );
```

4.13 ipac_write_ushort

NAME

ipac_write_ushort – Write one word (16-bit) to an IPAC space

SYNOPSIS

void ipac_write_ushort(struct addr_space_desc *space, unsigned long offset , unsigned short value);

DESCRIPTION

Write one word (16-bit) to the IPAC space location specified by the address space descriptor and the relative offset.

This access function always performs a big-endian access. Depending on the CPU architecture and carrier board hardware byte lanes will be swapped as necessary.

On PowerPC boards the access will be ordered (EIEIO).

PARAMETER

space

Address space descriptor pointer.

offset

Address offset (bytes) within this space.

value

Data word (16-bit) to write to the specified location.

EXAMPLE

```
#include ”ipac_carrier.h”

struct addr_space_desc *mem_space;

/* ... */

/* write 0xaa55 to offset 0x1000 within the MEM space */
ipac_write_ushort( io_space, 0x1000, 0xaa55 );
```

4.14 ipac_write_ulong

NAME

ipac_write_ulong – Write one long word (32-bit) to an IPAC space

SYNOPSIS

void ipac_write_ulong(struct addr_space_desc *space, unsigned long offset , unsigned long value);

DESCRIPTION

Write one long word (32-bit) to the IPAC space location specified by the address space descriptor and the relative offset.

This access function always performs a big-endian access. Depending on the CPU architecture and carrier board hardware word byte and word lanes will be swapped as necessary.

On PowerPC boards the access will be ordered (EIEIO).

PARAMETER

space

Address space descriptor pointer.

offset

Address offset (bytes) within this space.

value

Data long word (32-bit) to write to the specified location.

EXAMPLE

```
#include ”ipac_carrier.h”

struct addr_space_desc *mem_space;

/* ... */

/* write 0xdadada55 to offset 0x1000 within the MEM space */
ipac_write_ulong( mem_space, 0x1000, 0xdadada55 );
```

4.15 ipCarrierShow

NAME

ipCarrierShow – Show the contents of IPAC carrier data base

SYNOPSIS

void ipCarrierShow()

DESCRIPTION

This function can be used for debugging purposes to display the contents of the internal IPAC carrier data base. Usually this function is called at the VxWorks target shell to get information on found carrier boards, IPAC modules and slot resources (e.g. to access IPAC spaces manually).

EXAMPLE

```
-> ipCarrierInit
value = 0 = 0x0

-> ipCarrierShow

TVME8240 carrier with 4 slots found @ PCI bus=0, device=16
 - Slot[0]: IO=0xF5000000, ID=0xF5000080, INT=0xF50000C0, MEM8=0xF4000000,
MEM16=0xF2000000, IVEC=0x34, INT0_LVL=0
x34, INT1_LVL=0x34
            +++ Valid IP module mounted (Manufacturer=0xB3, Model=0x22)
 - Slot[1]: IO=0xF5000100, ID=0xF5000180, INT=0xF50001C0, MEM8=0xF4400000,
MEM16=0xF2800000, IVEC=0x34, INT0_LVL=0
x34, INT1_LVL=0x34
            --- NO or BAD IP module
 - Slot[2]: IO=0xF5000200, ID=0xF5000280, INT=0xF50002C0, MEM8=0xF4800000,
MEM16=0xF3000000, IVEC=0x34, INT0_LVL=0
x34, INT1_LVL=0x34
            --- NO or BAD IP module
 - Slot[3]: IO=0xF5000300, ID=0xF5000380, INT=0xF50003C0, MEM8=0xF4C00000,
MEM16=0xF3800000, IVEC=0x34, INT0_LVL=0
x34, INT1_LVL=0x34
            +++ Valid IP module mounted (Manufacturer=0xB3, Model=0x1C)
```

```
VMEbus carrier with 4 slots found
 - Slot[0]: IO=0xF1FF6000, ID=0xF1FF6080, INT=0x00000000, MEM8=0xF0D00000,
MEM16=0xF0D00000, IVEC=0xa0, INT0_LVL=0
x1, INT1_LVL=0x2
            --- NO or BAD IP module
 - Slot[1]: IO=0xF1FF6100, ID=0xF1FF6180, INT=0x00000000, MEM8=0xF0D40000,
MEM16=0xF0D40000, IVEC=0xa4, INT0_LVL=0
x3, INT1_LVL=0x4
            --- NO or BAD IP module
 - Slot[2]: IO=0xF1FF6200, ID=0xF1FF6280, INT=0x00000000, MEM8=0xF0D80000,
MEM16=0xF0D80000, IVEC=0xa8, INT0_LVL=0
x5, INT1_LVL=0x6
            --- NO or BAD IP module
 - Slot[3]: IO=0xF1FF6300, ID=0xF1FF6380, INT=0x00000000, MEM8=0xF0DC0000,
MEM16=0xF0DC0000, IVEC=0xac, INT0_LVL=0
x7, INT1_LVL=0x0
            --- NO or BAD IP module
```